EigenLabs — EigenLayer

Date	March 2023
Auditors	Heiko Fisch, Dominik Muhs

1 Executive Summary

This report presents the results of our engagement with **EigenLabs** to review a subset of their **EigenLayer** smart contracts, a system that enables restaking of assets to secure new services.

The review was conducted over three weeks, from **March 22, 2023**, to **April 11, 2023**, by **Dominik Muhs** and **Heiko Fisch**. A total of 30 person-days were spent.

During the first week, we familiarized ourselves with the in-scope components of the system and studied the exhaustive EigenLayer documentation. Basic checks regarding security best practices and the implementation's consistency were performed.

During the second week, we focused on individual components and internal functions such as transfers of ETH, contract interdependencies, and potential business logic vulnerabilities.

During the third week, we focused on business logic contained in libraries, functions of concentrated risk, and summarizing the audit's findings.

It is worth noting that while the `StrategyBase` contract is in scope, its expressiveness and security rely on user-defined strategy behavior, which is not yet present at the time of this engagement. Since isolated contracts in the inheritance hierarchy do not guarantee the eventual component's behavior, we advise taking the audit results regarding strategies and their related functionalities with a grain of salt. We recommend incorporating rigorous review processes on EigenLayer's strategies to minimize the risk introduced by potentially malicious or faulty contracts or contract upgrades.

2 Scope

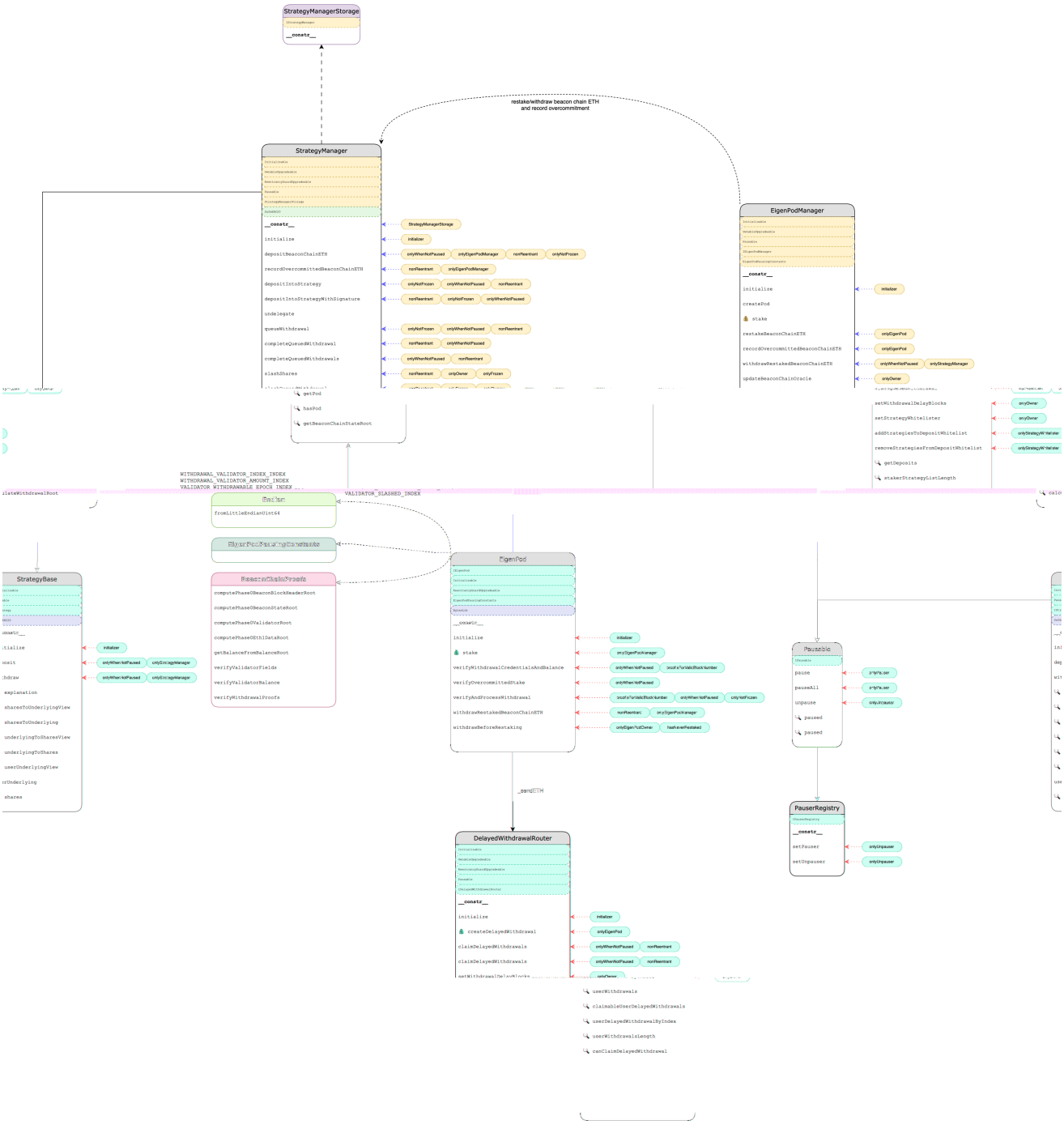
Our review focused on the commit hash 0aa1dc7b65f3a7a7b718d6c8bd847612cbf2ae08. The list of files in scope can be found in the Appendix.

2.1 Objectives

Together with the **EigenLayer** product team, we identified the following priorities for our review:

1. Correctness of the implementation, consistent with the intended functionality and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.
3. Any attack vectors resulting in the loss of user funds or actions defeating safeguards protecting user funds.
4. Validate the correctness of the native restaking flow.
5. Identify potential privilege escalation points across roles and components in the in-scope system.

3 System Overview



4 Findings

Each issue has an assigned severity:

- issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- issues are directly exploitable security vulnerabilities that need to be fixed.

4.1 Potential Reentrancy Into Strategies

The `StrategyManager` contract is the entry point for deposits into and withdrawals from strategies. More specifically, to deposit into a strategy, a staker calls `depositIntoStrategy` (or anyone calls `depositIntoStrategyWithSignature` with the staker's signature) then the asset is transferred from the staker to the strategy contract. After that, the strategy's `deposit` function is called, followed by some bookkeeping in the `StrategyManager`. For withdrawals (and slashing), the `StrategyManager` calls the strategy's `withdraw` function, which transfers the given amount of the asset to the given recipient. Both token transfers are a potential source of reentrancy if the token allows it.

The `StrategyManager` uses OpenZeppelin's `ReentrancyGuardUpgradeable` as reentrancy protection, and the relevant functions have a `nonReentrant` modifier. The `StrategyBase` contract – from which concrete strategies should be derived – does not have reentrancy protection. However, the functions `deposit` and `withdraw` can only be called from the `StrategyManager`, so reentering these is impossible.

Nevertheless, other functions could be reentered, for example, `sharesToUnderlyingView` and `underlyingToSharesView`, as well as their (supposedly) non-`view` counterparts.

Let's look at the `withdraw` function in `StrategyBase`. First, the `amountShares` shares are burnt, and at the end of the function, the equivalent amount of `token` is transferred to the `depositor`:

src/contracts/strategies/StrategyBase.sol:L108-L143

```
function withdraw(address depositor, IERC20 token, uint256 amountShares)
    external
    virtual
    override
    onlyWhenNotPaused(PAUSED_WITHDRAWALS)
    onlyStrategyManager
{
    require(token == underlyingToken, "StrategyBase.withdraw: Can only withdraw the strategy token");
    // copy `totalShares` value to memory, prior to any decrease
    uint256 priorTotalShares = totalShares;
    require(
        amountShares <= priorTotalShares,
        "StrategyBase.withdraw: amountShares must be less than or equal to totalShares"
    );

    // Calculate the value that `totalShares` will decrease to as a result of the withdrawal
    uint256 updatedTotalShares = priorTotalShares - amountShares;
    // check to avoid edge case where share rate can be massively inflated as a 'griefing' sort of attack
    require(updatedTotalShares >= MIN_NONZERO_TOTAL_SHARES || updatedTotalShares == 0,
        "StrategyBase.withdraw: updated totalShares amount would be nonzero but below MIN_NONZERO_TOTAL_SHARES");
    // Actually decrease the `totalShares` value
    totalShares = updatedTotalShares;

    /**
     * @notice calculation of amountToSend *mirrors* `sharesToUnderlying(amountShares)`, but is different since the `totalShares` has already
     * been decremented. Specifically, notice how we use `priorTotalShares` here instead of `totalShares`.
     */
    uint256 amountToSend;
    if (priorTotalShares == amountShares) {
        amountToSend = _tokenBalance();
    } else {
        amountToSend = (_tokenBalance() * amountShares) / priorTotalShares;
    }

    underlyingToken.safeTransfer(depositor, amountToSend);
}
```

If we assume that the `token` contract has a callback to the recipient of the transfer *before* the actual balance changes take place, then the recipient could reenter the strategy contract, for example, in `sharesToUnderlyingView`:

src/contracts/strategies/StrategyBase.sol:L159-L165

```
function sharesToUnderlyingView(uint256 amountShares) public view virtual override returns (uint256) {
    if (totalShares == 0) {
        return amountShares;
    } else {
        return (_tokenBalance() * amountShares) / totalShares;
    }
}
```

The crucial point is: If the callback is executed *before* the actual balance change, then `sharesToUnderlyingView` will report a bad result because the shares have already been burnt. Still, the token balance has not been updated yet.

For deposits, the token transfer to the strategy happens first, and the shares are minted after that:

src/contracts/core/StrategyManager.sol:L643-L652

```
function _depositIntoStrategy(address depositor, IStrategy strategy, IERC20 token, uint256 amount)
    internal
    onlyStrategiesWhitelistedForDeposit(strategy)
    returns (uint256 shares)
{
    // transfer tokens from the sender to the strategy
    token.safeTransferFrom(msg.sender, address(strategy), amount);

    // deposit the assets into the specified strategy and get the equivalent amount of shares in that strategy
    shares = strategy.deposit(token, amount);
```

src/contracts/strategies/StrategyBase.sol:L69-L99

```
function deposit(IERC20 token, uint256 amount)
    external
    virtual
    override
    onlyWhenNotPaused(PAUSED_DEPOSITS)
    onlyStrategyManager
    returns (uint256 newShares)
{
    require(token == underlyingToken, "StrategyBase.deposit: Can only deposit underlyingToken");

    /**
     * @notice calculation of newShares *mirrors* `underlyingToShares(amount)`, but is different since the balance of `underlyingToken`
     * has already been increased due to the `strategyManager` transferring tokens to this strategy prior to calling this function
     */
    uint256 priorTokenBalance = _tokenBalance() - amount;
    if (priorTokenBalance == 0 || totalShares == 0) {
        newShares = amount;
    } else {
        newShares = (amount * totalShares) / priorTokenBalance;
    }

    // checks to ensure correctness / avoid edge case where share rate can be massively inflated as a 'griefing' sort of attack
    require(newShares != 0, "StrategyBase.deposit: newShares cannot be zero");
    uint256 updatedTotalShares = totalShares + newShares;
    require(updatedTotalShares >= MIN_NONZERO_TOTAL_SHARES,
        "StrategyBase.deposit: updated totalShares amount would be nonzero but below MIN_NONZERO_TOTAL_SHARES");

    // update total share amount
    totalShares = updatedTotalShares;
    return newShares;
}
```

That means if there is a callback in the token's `transferFrom` function and it is executed *after* the balance change, a reentering call to `sharesToUnderlyingView` (for example) will again return a wrong result because shares and token balances are not "in sync."

In addition to the reversed order of token transfer and shares update, there's another vital difference between `withdraw` and `deposit`: For withdrawals, the call to the token contract originates in the strategy, while for deposits, it is the strategy *manager* that initiates the call to the token contract (before calling into the strategy). That's a technicality that has consequences for reentrancy protection: Note that for withdrawals, it is the strategy contract that is reentered, while for deposits, there is not a single contract that is reentered; instead, it is the contract *system* that is in an inconsistent state when the reentrancy happens. Hence, reentrancy protection on the level of individual contracts is not sufficient.

Finally, we want to discuss though *which* functions in the strategy contract the system could be reentered. As mentioned, `deposit` and `withdraw` can only be called by the strategy manager, so these two can be ruled out. For the examples above, we considered `sharesToUnderlyingView`, which (as the name suggests) is a `view` function. As such, it can't change the state of the contract, so reentrancy through a `view` function can only be a problem for *other* contracts that use this function and rely on its return value. However, there is also a potentially state-changing variant, `sharesToUnderlying`, and similar potentially state-changing functions, such as `underlyingToShares` and `userUnderlying`. Currently, these functions are not *actually* state-changing, but the idea is that they could be and, in some concrete strategy implementations that inherit from `StrategyBase`, will be. In such cases, these functions could make wrong state changes due to state inconsistency during reentrancy.

The examples above assume that the token contract allows reentrancy through its `transfer` function *before* the balance change has been made or in its `transferFrom` function *after*. It might be tempting to argue that tokens which don't fall into this category are safe to use. While the examples discussed above are the most interesting attack vectors we found, there might still be others: To illustrate this point, assume a token contract that allows reentrancy through `transferFrom` only before any state change in the token takes place. The token transfer is the first thing that happens in `StrategyManager._depositIntoStrategy`, and the state changes (user shares) and calling the strategy's `deposit` function occur later, this might look safe. However, if the deposit happens via `StrategyManager.depositIntoStrategyWithSignature`, then it can be seen, for example, that the staker's nonce is updated before the internal `_depositIntoStrategy` function is called:

src/contracts/core/StrategyManager.sol:L244-L286

```
function depositIntoStrategyWithSignature(
    IStrategy strategy,
    IERC20 token,
    uint256 amount,
    address staker,
    uint256 expiry,
    bytes memory signature
)
    external
    onlyWhenNotPaused(PAUSED_DEPOSITS)
    onlyNotFrozen(staker)
    nonReentrant
    returns (uint256 shares)
{
    require(
        expiry >= block.timestamp,
        "StrategyManager.depositIntoStrategyWithSignature: signature expired"
    );
    // calculate struct hash, then increment `staker`'s nonce
    uint256 nonce = nonces[staker];
    bytes32 structHash = keccak256(abi.encode(DEPOSIT_TYPEHASH, strategy, token, amount, nonce, expiry));
    unchecked {
        nonces[staker] = nonce + 1;
    }
    bytes32 digestHash = keccak256(abi.encodePacked("\x19\x01", DOMAIN_SEPARATOR, structHash));


    /**
     * check validity of signature:
     * 1) if `staker` is an EOA, then `signature` must be a valid ECSDA signature from `staker`,
     * indicating their intention for this action
     * 2) if `staker` is a contract, then `signature` must will be checked according to EIP-1271
     */
    if (Address.isContract(staker)) {
        require(IERC1271(staker).isValidSignature(digestHash, signature) == ERC1271_MAGICVALUE,
            "StrategyManager.depositIntoStrategyWithSignature: ERC1271 signature verification failed");
    } else {
        require(ECDSA.recover(digestHash, signature) == staker,
            "StrategyManager.depositIntoStrategyWithSignature: signature not from staker");
    }

    shares = _depositIntoStrategy(staker, strategy, token, amount);
}
```

Hence, querying the staker's nonce in reentrancy would still give a result based on an "incomplete state change." It is, for example, conceivable that the staker still has zero shares, and yet their nonce is already 1. This particular situation is most likely not an issue, but the example shows that reentrancy can be subtle.

This is fine if the token doesn't allow reentrancy in the first place. As discussed above, among the tokens that do allow reentrancy, some variants of when reentrancy can happen in relation to state changes in the token seem more dangerous than others, but we have also argued that this kind of reasoning can be dangerous and error-prone. Hence, we recommend employing comprehensive and defensive reentrancy protection based on reentrancy guards such as OpenZeppelin's ReentrancyGuardUpgradeable, which is already used in the `StrategyManager`.

Unfortunately, securing a multi-contract system against reentrancy can be challenging, but we hope the preceding discussion and the following pointers will prove helpful:

1. External functions in strategies that should only be callable by the strategy manager (such as `deposit` and `withdraw`) should have the `onlyStrategyManager` modifier. This is already the case in the current codebase and is listed here only for completeness.
2. External functions in strategies for which item 1 doesn't apply (such as `sharesToUnderlying` and `underlyingToShares`) should query the strategy manager's reentrancy lock and revert if it is set.
3. In principle, the restrictions above also apply to `public` functions, but if a `public` function is also used internally, checks against reentrancy can cause problems (if used in an internal context) or at least be redundant. In the context of reentrancy protection, it is often easier to split public functions into an `internal` and an `external` one.
4. If `view` functions are supposed to give reliable results (either internally – which is typically the case – or for other contracts), they have to be protected too.
5. The previous item also applies to the `StrategyManager`: `view` functions that provide correct results should query the reentrancy lock and revert if it is set.
6. Solidity automatically generates getters for `public` state variables. Again, if these (`external view`) functions must deliver correct results, the same measures must be taken for explicit `view` functions. In practice, the state variable has to become `internal` or `private`, and the getter function must be hand-written.
7. The `StrategyBase` contract provides some basic functionality. Concrete strategy implementations can inherit from this contract, meaning that some functions may be overridden (and might or might not call the overridden version via `super`), and new functions might be added. While the guidelines above should be helpful, derived contracts must be reviewed and assessed separately on a case-by-case basis. As mentioned before, reentrancy protection can be challenging, especially in a multi-contract system.

4.2 `StrategyBase` – Inflation Attack Prevention Can Lead to Stuck Funds

As a defense against what has come to be known as inflation or donation attack in the context of ERC-4626, the `StrategyBase` contract – from which concrete strategy implementations are supposed to inherit – enforces that the amount of shares in existence for a particular strategy is always either 0 or at least a certain minimum amount that is set to 10^9. This mitigates inflation attacks, which require a small total supply of shares to be effective.

src/contracts/strategies/StrategyBase.sol:L92-L95

```
uint256 updatedTotalShares = totalShares + newShares;
require(updatedTotalShares >= MIN_NONZERO_TOTAL_SHARES,
    "StrategyBase.deposit: updated totalShares amount would be nonzero but below MIN_NONZERO_TOTAL_SHARES");
```

src/contracts/strategies/StrategyBase.sol:L123-L127

```
// Calculate the value that `totalShares` will decrease to as a result of the withdrawal
uint256 updatedTotalShares = priorTotalShares - amountShares;
// check to avoid edge case where share rate can be massively inflated as a 'griefing' sort of attack
require(updatedTotalShares >= MIN_NONZERO_TOTAL_SHARES || updatedTotalShares == 0,
    "StrategyBase.withdraw: updated totalShares amount would be nonzero but below MIN_NONZERO_TOTAL_SHARES");
```

This particular approach has the downside that, in the worst case, a user may be unable to withdraw the underlying asset for up to 10^9 - 1 shares. While the extreme circumstances under which this can happen might be unlikely to occur in a realistic setting and, in many cases, the value of 10^9 - 1 shares may be negligible, this is not ideal.

It isn't easy to give a good general recommendation. None of the suggested mitigations are without a downside, and what's the best choice may also depend on the specific situation. We do, however, feel that alternative approaches that can't lead to stuck funds might be worth considering, especially for a default implementation.

One option is internal accounting, i.e., the strategy keeps track of the number of underlying tokens it owns. It uses this number for conversion rate calculation instead of its balance in the token contract. This avoids the donation attack because sending tokens directly to the strategy will not affect the conversion rate. Moreover, this technique helps prevent reentrancy issues when the EigenLayer state is out of sync with the token contract's state. The downside is higher gas costs and that donating by just sending tokens to the contract is impossible; more specifically, if it happens accidentally, the funds are lost unless there's some special mechanism to recover them.

An alternative approach with virtual shares and assets is presented here, and the document lists pointers to more discussions and proposed solutions.

4.3 `StrategyWrapper` – Functions Shouldn't Be `virtual` (Out of Scope)

The `StrategyWrapper` contract is a straightforward strategy implementation and – as its NatSpec documentation explicitly states – is not designed to be inherited from:

src/contracts/strategies/StrategyWrapper.sol:L8-L17

```
/**
 * @title Extremely simple implementation of `IStrategy` interface.
 * @author Layr Labs, Inc.
 * @notice Simple, basic, "do-nothing" Strategy that holds a single underlying token and returns it on withdrawals.
 * Assumes shares are always 1-to-1 with the underlyingToken.
 * @dev Unlike `StrategyBase`, this contract is *not* designed to be inherited from.
 * @dev This contract is expressly *not* intended for use with 'fee-on-transfer'-type tokens.
 * Setting the `underlyingToken` to be a fee-on-transfer token may result in improper accounting.
 */
contract StrategyWrapper is IStrategy {
```

However, all functions in this contract are `virtual`, which only makes sense if inheriting from `StrategyWrapper` is possible.

Assuming the NatSpec documentation is correct, and no contract should inherit from `StrategyWrapper`, remove the `virtual` keyword from all function definitions. Otherwise, fix the documentation.

This contract is out of scope, and this finding is only included because we noticed it accidentally. This does not mean we have reviewed the contract or other out-of-scope files.

4.4 `StrategyBase` – Inheritance-Related Issues

A. The `StrategyBase` contract defines `view` functions that, given an amount of shares, return the equivalent amount of tokens (`sharesToUnderlyingView`) and vice versa (`underlyingToSharesView`). These two functions also have non-`view` counterparts: `sharesToUnderlying` and `underlyingToShares`, and their NatSpec documentation explicitly states that they should be allowed to make state changes. Given the scope of this engagement, it is unclear if these non-`view` versions are needed, but assuming they are, this does currently not work as intended.

First, the interface `IStrategy` declares `underlyingToShares` as `view` (unlike `sharesToUnderlying`). This means overriding this function in derived contracts is impossible without the `view` modifier. Hence, in `StrategyBase` – which implements the `IStrategy` interface – this (virtual) function is (and has to be) `view`. The same applies to overridden versions of this function in contracts inherited from `StrategyBase`.

src/contracts/interfaces/IStrategy.sol:L39-L45

```
/**
 * @notice Used to convert an amount of underlying tokens to the equivalent amount of shares in this strategy.
 * @notice In contrast to `underlyingToSharesView`, this function **may** make state modifications
 * @param amountUnderlying is the amount of `underlyingToken` to calculate its conversion into strategy shares
 * @dev Implementation for these functions in particular may vary signifcantly for different strategies
 */
function underlyingToShares(uint256 amountUnderlying) external view returns (uint256);
```

src/contracts/strategies/StrategyBase.sol:L192-L200

```
/**
 * @notice Used to convert an amount of underlying tokens to the equivalent amount of shares in this strategy.
 * @notice In contrast to `underlyingToSharesView`, this function **may** make state modifications
 * @param amountUnderlying is the amount of `underlyingToken` to calculate its conversion into strategy shares
 * @dev Implementation for these functions in particular may vary signifcantly for different strategies
 */
function underlyingToShares(uint256 amountUnderlying) external view virtual returns (uint256) {
    return underlyingToSharesView(amountUnderlying);
}
```

As mentioned above, the `sharesToUnderlying` function does not have the `view` modifier in the interface `IStrategy`. However, the overridden (and virtual) version in `StrategyBase` does, which means again that overriding this function in contracts inherited from `StrategyBase` is impossible without the `view` modifier.

src/contracts/strategies/StrategyBase.sol:L167-L175

```
/**
 * @notice Used to convert a number of shares to the equivalent amount of underlying tokens for this strategy.
 * @notice In contrast to `sharesToUnderlyingView`, this function **may** make state modifications
 * @param amountShares is the amount of shares to calculate its conversion into the underlying token
 * @dev Implementation for these functions in particular may vary signifcantly for different strategies
 */
function sharesToUnderlying(uint256 amountShares) public view virtual override returns (uint256) {
    return sharesToUnderlyingView(amountShares);
}
```

B. The `initialize` function in the `StrategyBase` contract is not virtual, which means the name will not be available in derived contracts (unless with different parameter types). It also has the `initializer` modifier, which is unavailable in concrete strategies inherited from `StrategyBase`.

A. If state-changing versions of the conversion functions are needed, the `view` modifier has to be removed from `IStrategy.underlyingToShares`, `StrategyBase.underlyingToShares`, and `StrategyBase.sharesToUnderlying`. They should be removed entirely from the interface and base contract if they're not needed.

B. Consider making the `StrategyBase` contract `abstract`, maybe give the `initialize` function a more specific name such as `_initializeStrategyBase`, change its visibility to `internal`, and use the `onlyInitializing` modifier instead of `initializer`.

4.5 `StrategyManager` - Cross-Chain Replay Attacks After Chain Split Due to Hard-Coded `DOMAIN_SEPARATOR`

A. The `StrategyManager` contract allows stakers to deposit into and withdraw from strategies. A staker can either deposit themself or have someone else do it on their behalf, where the latter requires an EIP-712-compliant signature. The EIP-712 domain separator is computed in the `initialize` function and stored in a state variable for later retrieval:

src/contracts/core/StrategyManagerStorage.sol:L23-L24

```
/// @notice EIP-712 Domain separator
bytes32 public DOMAIN_SEPARATOR;
```

src/contracts/core/StrategyManager.sol:L149-L153

```
function initialize(address initialOwner, address initialStrategyWhitelister, IPauserRegistry _pauserRegistry, uint256 initialPausedStatus, uint256 _withdrawalDelayBlocks)
    external
    initializer
{
    DOMAIN_SEPARATOR = keccak256(abi.encode(DOMAIN_TYPEHASH, bytes("EigenLayer"), block.chainid, address(this)));
```

Once set in the `initialize` function, the value can't be changed anymore. In particular, the chain ID is "baked into" the `DOMAIN_SEPARATOR` during initialization. However, it is not necessarily constant: In the event of a chain split, only one of the resulting chains gets to keep the original chain ID, and the other should use a new one. With the current approach to compute the `DOMAIN_SEPARATOR` during initialization, store it, and then use the stored value for signature verification, a signature will be valid on both chains after a split – but it should not be valid on the chain with the new ID. Hence, the domain separator should be computed dynamically.

B. The `name` in the `EIP712Domain` is of type `string`:

src/contracts/core/StrategyManagerStorage.sol:L18-L19

```
bytes32 public constant DOMAIN_TYPEHASH =
    keccak256("EIP712Domain(string name,uint256 chainId,address verifyingContract)");
```

What's encoded when the domain separator is computed is `bytes("EigenLayer")`:

src/contracts/core/StrategyManager.sol:L153

```
DOMAIN_SEPARATOR = keccak256(abi.encode(DOMAIN_TYPEHASH, bytes("EigenLayer"), block.chainid, address(this)));
```

According to EIP-712,

> The dynamic values `bytes` and `string` are encoded as a `keccak256` hash of their contents.

Hence, `bytes("EigenLayer")` should be replaced with `keccak256(bytes("EigenLayer"))`.

C. The `EIP712Domain` does not include a version string:

src/contracts/core/StrategyManagerStorage.sol:L18-L19

```
bytes32 public constant DOMAIN_TYPEHASH =
    keccak256("EIP712Domain(string name,uint256 chainId,address verifyingContract)");
```

That is allowed according to the specification. However, given that most, if not all, projects, as well as OpenZeppelin's EIP-712 implementation, do include a version string in their `EIP712Domain`, it might be a pragmatic choice to do the same, perhaps to avoid potential incompatibilities.

Individual recommendations have been given above. Alternatively, you might want to utilize OpenZeppelin's `EIP712Upgradeable` library, which will take care of these issues. **Note that some of these changes will break existing signatures.**

4.6 `StrategyManagerStorage` – Miscalculated Gap Size

Upgradeable contracts should have a "gap" of unused storage slots at the end to allow for adding state variables when the contract is upgraded. The convention is to have a gap whose size adds up to 50 with the used slots at the beginning of the contract's storage.

In `StrategyManagerStorage`, the number of consecutively used storage slots is 10:

- `DOMAIN_SEPARATOR`
- `nonces`
- `strategyWhitelister`
- `withdrawalDelayBlocks`
- `stakerStrategyShares`
- `stakerStrategyList`
- `withdrawalRootPending`
- `numWithdrawalsQueued`
- `strategyIsWhitelistedForDeposit`
- `beaconChainETHSharesToDecrementOnWithdrawal`

```
/**
 * @notice Called in order to withdraw delayed withdrawals made to the caller that have passed the `withdrawalDelayBlocks` period.
 * @param maxNumberOfDelayedWithdrawalsToClaim Used to limit the maximum number of delayedWithdrawals to loop through claiming.
 */
function claimDelayedWithdrawals(uint256 maxNumberOfDelayedWithdrawalsToClaim)
    external
    nonReentrant
    onlyWhenNotPaused(PAUSED_DELAYED_WITHDRAWAL_CLAIMS)
{
    _claimDelayedWithdrawals(msg.sender, maxNumberOfDelayedWithdrawalsToClaim);
}
```

src/contracts/pods/DelayedWithdrawalRouter.sol:L71-L82

```
/**
 * @notice Called in order to withdraw delayed withdrawals made to the `recipient` that have passed the `withdrawalDelayBlocks` period.
 * @param recipient The address to claim delayedWithdrawals for.
 * @param maxNumberOfDelayedWithdrawalsToClaim Used to limit the maximum number of delayedWithdrawals to loop through claiming.
 */
function claimDelayedWithdrawals(address recipient, uint256 maxNumberOfDelayedWithdrawalsToClaim)
    external
    nonReentrant
    onlyWhenNotPaused(PAUSED_DELAYED_WITHDRAWAL_CLAIMS)
{
    _claimDelayedWithdrawals(recipient, maxNumberOfDelayedWithdrawalsToClaim);
}
```

An attacker can not control *where* the funds are sent, but they can control *when* the funds are sent once the withdrawal becomes claimable. It is unclear whether this can become a problem. It is, for example, conceivable that there are negative tax implications if funds arrive earlier than intended at a particular address: For instance, disadvantages for the recipient can arise if funds arrive before the new year starts or before some other funds were sent to the same address (e.g., in case the FIFO principle is applied for taxes).

The downside of allowing only the recipient to claim their withdrawals is that contract recipients must be equipped to make the claim.

If these points haven't been considered yet, we recommend doing so.

4.9 Consider Using Custom Errors

Custom errors were introduced in Solidity version 0.8.4 and have some advantages over the traditional string-based errors: They are usually more gas-efficient, especially regarding deployment costs, and it is easier to include dynamic information in error messages.

4.10 `StrategyManager` - Immediate Settings Changes Can Have Unintended Side Effects

Withdrawal delay blocks can be changed immediately:

src/contracts/core/StrategyManager.sol:L570-L572

```
function setWithdrawalDelayBlocks(uint256 _withdrawalDelayBlocks) external onlyOwner {
    _setWithdrawalDelayBlocks(_withdrawalDelayBlocks);
}
```

Allows owner to sandwich e.g. `completeQueuedWithdrawal` function calls to prevent users from withdrawing their stake due to the following check:

src/contracts/core/StrategyManager.sol:L749-L753

```
require(queuedWithdrawal.withdrawalStartBlock + withdrawalDelayBlocks <= block.number
        || queuedWithdrawal.strategies[0] == beaconChainETHStrategy,
    "StrategyManager.completeQueuedWithdrawal: wi 0 0 4llDelayBlock perios hsn no yete passe",
```

The modifier can be removed.

4.12 Inconsistent Data Types for Block Numbers

The block number attribute is declared using `uint32` and `uint64`. This usage should be more consistent.

uint32

src/contracts/interfaces/IEigenPod.sol:L30-L35

```
struct PartialWithdrawalClaim {
    PARTIAL_WITHDRAWAL_CLAIM_STATUS status;
    // block at which the PartialWithdrawalClaim was created
    uint32 creationBlockNumber;
    // last block (inclusive) in which the PartialWithdrawalClaim can be fraudproofed
    uint32 fraudproofPeriodEndBlockNumber;
```

src/contracts/core/StrategyManager.sol:L393

```
withdrawalStartBlock: uint32(block.number),
```

src/contracts/pods/DelayedWithdrawalRouter.sol:L62-L65

```
DelayedWithdrawal memory delayedWithdrawal = DelayedWithdrawal({
    amount: withdrawalAmount,
    blockCreated: uint32(block.number)
});
```

uint64

src/contracts/pods/EigenPod.sol:L175-L176

```
function verifyWithdrawalCredentialsAndBalance(
    uint64 oracleBlockNumber,
```

src/contracts/pods/EigenPodManager.sol:L216

```
function getBeaconChainStateRoot(uint64 blockNumber) external view returns(bytes32) {
```

Use one data type consistently to minimize the risk of conversion errors or truncation during casts. This is a measure aimed at future-proofing the code base.

4.13 `EigenPod` – Stray `nonReentrant` Modifier

There is a stray `nonReentrant` modifier in `EigenPod`:

src/contracts/pods/EigenPod.sol:L432-L453

```
/**
 * @notice Transfers `amountWei` in ether from this contract to the specified `recipient` address
 * @notice Called by EigenPodManager to withdrawBeaconChainETH that has been added to the EigenPod's balance due to a withdrawal from the beacon chain.
 * @dev Called during withdrawal or slashing.
 * @dev Note that this function is marked as non-reentrant to prevent the recipient calling back into it
 */
function withdrawRestakedBeaconChainETH(
    address recipient,
    uint256 amountWei
)
    external
    onlyEigenPodManager
    nonReentrant
{
    // reduce the restakedExecutionLayerGwei
    restakedExecutionLayerGwei -= uint64(amountWei / GWEI_TO_WEI);

    emit RestakedBeaconChainETHWithdrawn(recipient, amountWei);

    // transfer ETH from pod to `recipient`
    _sendETH(recipient, amountWei);
}
```

src/contracts/pods/EigenPod.sol:L466-L468

Request a Security Review Today

Get in touch with our team to request a quote for a smart contract audit.

Subscribe to Our Newsletter

Stay up-to-date on our latest offerings, tools, and the world of blockchain security.